MOAB has unit tests there can be run by executing 'make check'. Use the instructions below to add a new unit test for MOAB. If your tests rely on any input files or result in the generation of any output files, make sure those are handled properly too.

The best way to develop a new unit test is to start with an existing one and modify it to suit your needs. But, you should also be able to develop one from scratch using the instructions below.

Adding a new unit test (simple version)

1. Write your test in the form of an executable that returns the number of failures, with zero indicating success. Store this file in the test/ subdirectory, or, if it's associated with input/output, in the io subdirectory below that.
2. Edit the Makefile.am file in the same directory as your test file:
    1. Add the name of your executable, e.g. <my_test>, to the TESTS variable
    2. Add a line further below indicating the source file, e.g. <my_test>_SOURCES = <my_test>.cpp
3. Try running your test, by executing 'make', then 'make check'. You should see something like:

```
PASS: <my_test>
```

if your test passes.

Utilities for Writing Tests

The TestUtil.hpp file in the test/ subdirectory of the MOAB source provides a lightweight testing framework. Its primary features are that a) it facilitates decomposing the various features that are tested in a single test executable into separate sub-tests, making maintenance of the test easier as well as making it easier to see at a glance which feature is failing and b) it provides utility macros to assist with reporting test failure diagnostics.

Each sub-test is defined as a function of type `void(*)()` (a function with no arguments and no return type). The `main` routine of the test executable then calls the `RUN_TEST(function)` macro for each sub-test. The macro evaluates to zero if the test succeeds and one if it fails, so the total number of failed tests can be calculated and returned from the `main` routine by summing the results of each `RUN_TEST` invocation.

The framework does not provide a mechanism to report a test failure directly. Rather, it provides a series of macros to test conditions that must be true if the test has succeeded. Examples include:

- `CHECK(condition)` - Test fails if condition is not true
- `CHECK_ERR(errorcode)` - Test fails if `errorcode` is not `MB_SUCCESS`
- `CHECK_EQUAL(A,B)` - Test fails if passed values are not equal.

See TestUtil.hpp for a complete list of macros. Using the most specific `CHECK_*` macro possible will result in the most informative output for test failues (for example, prefer `CHECK_EQUAL(5,count)` to `CHECK(count == 5)`.) Also, note that all `CHECK_*` macros that compare two values will report the first value as the expected value and the second argument as the actual value.

For a simple example of using these utilities, see test/io/read_cgm_test.cpp.

The TestRunner.hpp file defines a higher-level interface for running tests. It is built on top of the functionality provided in TestUtil.hpp. It provides most of the body of a `main` routine that uses the TestUtil.hpp framework with

the additional features of supporting dependent tests (tests that are not run if some other required test fails) and allowing specific sub-tests to run to be specified on the command line of the test executable. To use this framework, specify each test that is to be run by calling the `REGISTER_TEST` or `REGISTER_DEP_TEST` macros for each test function from the `main` routine, and then return the result of a call to the `RUN_TESTS(argc,argv)` macro from the `main` routine.

See test/test_prog_opt.cpp for a demonstration of the use of TestRunner.hpp .

Tests that require input files

Some tests require input files. The relative path to these files is different for in-source vs. out-of-source builds. To handle this, use the following construct to construct the fully-qualified filename:

```
#ifdef MESHDIR
static const char my_file[] = STRINGIFY(MESHDIR) "/io/my_file.h5m";
#else
static const char my_file[] = "my_file.h5m";
#endif
```

The value of MESHDIR is set to the location of the MeshFiles/unittest directory under the MOAB source, so in the above example the input file would be located in MeshFiles/unittest/io/my_file.h5m.

Tests that write output files

If your test writes an output file, that file should be cleaned up as a result of 'make clean'. This is accomplished by adding the filename of any generated file to the MOSTLYCLEANFILES variable in the Makefile.am in the directory in which the file is written (usually the same directory as your test executable).